View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 4 (2011) 332–341

Procedia Computer Science

International Conference on Computational Science, ICCS 2011

Query-driven Multiscale Data Postprocessing in Computational Fluid Dynamics

Atanas Atanasov[1,*], Tobias Weinzierl[**]

Scientific Computing in Computer Science, Institut für Informatik, Technische Universität München

Abstract

Massively parallel computational fluid dynamics codes that have to stream solution data to a visualisation or postprocessing component in each time step often are IO-bounded. This is especially cumbersome if the succeeding components require the simulation data only in a coarse resolution or only in specific subregions. We suggest to replace the streaming data approach found in many applications with a query-driven communication paradigm where the postprocessing components explicitly inform the fluid solver which data they need in which resolution in which subregions. Two case studies reveal that such a data exchange paradigm reduces the memory footprint of the exchanged data as well as the latency of the data delivery, and that the approach scales. In particular geometric multigrid solvers based upon a non-overlapping domain decomposition can answer such queries efficiently.

Keywords: Computational steering, Computational fluid mechanics, Multiscale methods, Domain decomposition, Problem solving environments

1. Introduction

The computer science challenges in computational fluid dynamics (CFD) nowadays comprise, besides the pure simulation tasks, software integration aspects, too. CFD codes have to fit into the scientific application landscape. They become one component in a zoo of components, i.e. their output data is postprocessed by multiple codes and they also react to input. These data exchange characteristics describe one fundamental challenge of computational steering and visual supercomputing [1]. In this context, we analyse two case studies highlighting shortcomings of classical data flow paradigms: flow through a porous media, and a transient flow which is used as input data for an external tool tracking particles suspended in the medium. Both applications have three properties in common: To track the solution in time, we need an on-the-fly visualisation of the fluid field on a rather coarse level of detail. To monitor and postprocess the regions of particular interest—narrow passages in the porous medium or regions around the particles—we need a very detailed representation of the velocities and the pressure there. Finally, these regions change in time.

*atanasoa@in.tum.de

**weinzier@in.tum.de

[1]Corresponding author

1877–0509 © 2011 Published by Elsevier Ltd. Open access under CC BY-NC-ND license.

Selection and/or peer-review under responsibility of Prof. Mitsuhisa Sato and Prof. Satoshi Matsuoka

doi:10.1016/j.procs.2011.04.035

Many CFD applications write the whole simulation data—simulation snapshots—to a stream. Remote applications then interpret the data through appropriate postprocessing techniques. We found this classical approach inadequate due to bandwidth restrictions. This is particularly annoying, as most of the data is not required in full resolution. We found a preprocessing step on the CFD side which reduces the amount of data inadequate, as our consecutive codes need the data in different resolutions in different regions and these regions are time-dependent and not known a priori. We found storing data on a medium attached to the supercomputer inadequate as the data size is enormous. We found streaming the dynamically adaptive grids of the CFD solver inadequate, as our visualisation and postprocessing algorithms were written for regular Cartesian grids.

In this paper, we propose a two-fold strategy that overcomes these drawbacks. As we switch from a producer-consumer paradigm to a client-server architecture, the fluid solver acts as server for the consecutive codes. And as we switch from fire-and-forget semantics to an on-demand paradigm, the CFD code delivers only data really required by the postprocessing steps, i.e. we "invest" bandwidth where it does the most good. The key ingredient here is a query language: the postprocessing codes pass queries to the fluid solver. These queries comprise the spatial region of interest, the resolution, and the variable of interest. The solver then returns a Cartesian grid of the requested data. This is a low-overhead data structure well-suited for our postprocessing needs (e.g. well-suited for textures of a GPU). Besides the simplicity of the data structures, our server implementation also exploits the multiscale nature of the parallel solver to react to the queries fast. A combination of these two ideas reduces the bandwidth required, as solely data needed is streamed to other codes, the response time scales on a parallel computer due to the structuredness and simplicity of the data streamed, the answer latency is small due to the asynchronous answering where the multiscale solver uses already coarsed data, and the multilevel and adaptive grids from the CFD code are hidden from the postprocessing units, i.e. the solver neither is to be tailored to consecutive steps nor do the postprocessing codes know the solver's realisation.

How to postprocess simulation data on a supercomputer efficiently is a challenge gaining impact due to the massive rise of cores on supercomputers. At least three trends are studied extensively: First, in-situ postprocessing where the postprocessing is done directly on the computing node [1, 2]. Such an approach is suitable particularly on heterogeneous environments with graphics cards physically attached to the computing node as postprocessing devices, and it streams only data really studied to the user. Second, in-situ data conversion where the data on the supercomputing is converted into a format fitting to the postprocessing needs such as texture maps [3], into multiscale representations [4], or is compressed such as proposed in [5] or indexed by well-suited keys [6]. Third, IO forwarding where the slow IO operations are deployed to specialised cores [7]—an idea integrating pervasive parallelism into the data flow architecture.

Our approach is orthogonal and might be combined with all three trends to tackle the data exploration challenge holistically as postulated by [8]. While it does some in-situ postprocessing, it however differs from many in-situ approaches as it does not deploy all the postprocessing tasks to postprocessing cores or cards. Instead, it already reduces the data streamed to these devices. In this context, it is a complex stream processing on the data source node for scientific computing. Simulation data is filtered according to prescribed needs, and only these data is streamed to the postprocessing nodes. The computing nodes are not a sole data source but the supercomputer becomes a demand-driven data server similar to [9] that supports multiresolution remote rendering [4]. In return, the delivered data exhibits a low memory overhead and is perfectly suited for texture-based postprocessing [3], i.e. can take benefit of GPU cards attached to the supercomputing node. The combination of several queries with different resolutions facilitates AMR-like postprocessing [10]. In the following, we put these integration aspects from end-to-end supercomputing aside and neglect sophisticated in-situ postprocessing. Consequently, our idea of a query and data of interest comprises solely spatial information and, different to database aligned approaches, does not take into account the semantics of the data such as upper and lower thresholds—techniques referred to as query-driven visualisation [6].

The remainder is organised as follows: We first introduce our two application examples. In Section 3, we then present our query language and, hence, define the data exchange paradigm, before we give some details on the realisation in Section 4. Some results (Section 5) and a short conclusion close the discussion.

2. Application Examples

2.1. Flow through porous media

The first application demonstrator is a scenario from computational fluid dynamics where a chemical substance pervades a porous rock filled with water. As the rock's conductivity is neglectable, the substance spreads solely in the water. While the diffusion equation underlying this experiment follows a state-of-the-art finite element formulation being sufficiently robust and accurate, the representation and handling of the porous media is difficult. Due to a lack of microscale data, we synthetically generate porous media geometries resulting from a combination of spheres (sphere package) and other geometric primitives, and tailor these artificial geometries manually to make it fit to real measurements. Passed to the finite element solver, such grids tend to induce geometric locking where the clearance between rock obstacles is blocked due to an insufficient tessellation accuracy (Figure 1). While some lockings might resemble the real world where the rocks glue to their neighbours, others are physically incorrect and not in accordance with the imitated and measured real-world flow. Refining the grid in-between the rocks or slight manipulations of the rock surface shape then put things right.

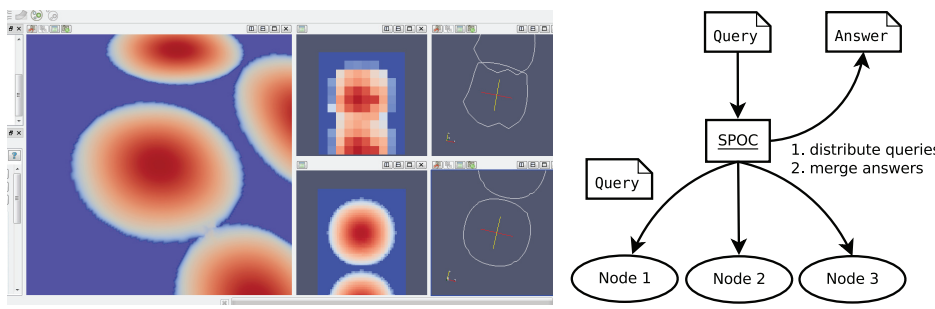

Figure 1: The discretised model introduces unphysical impenetrable regions for the fluid (left); user zooms into these regions interactively while the simulation runs to analyse whether the experimental setup has to be modified or whether there is a really an obstacle. To zoom into the data the user posts a query to a single point of contact, which represents a parallel server and waits for the merged answer(right). As a result, the user guides the grid to refine some regions further to avoid the locking.

Our vision of a simulation workflow reads as follows: We provide the user with a very coarse on-the-fly visualisation of the diffusion process. Throughout the simulation, the user then identifies regions with unphysically lockings, zooms into these regions, analyses them, and modifies the discretisation. Multiple regions of special interest might exist, and regions visualised with multiple level of details might overlap. The geometry manipulations in this computational steering scenario are facilitated by the remeshing feature of our CFD code, so the challenge is to give the user the opportunity to find and study the locking regions while the simulation runs on the supercomputer.

2.2. Drift Ratchet

The second application demonstrator is a scenario from microscale particle transportation studies [11]. Here, water swaps forth and back through very small asymmetric pores. Particles are suspended in the water and swap forth and back, too. However, their runtime behaviour differs from the periodic water flow, as the Brownian motion of the fluid molecules induces some long-term particle drift along or versus the water oscillation direction.

Our simulation exhibits three processing stages where we do not compute a classical fluid-structure interaction but work with virtual particles which are not taken into account by the CFD code. First, we simulate the water movement without any particle. Second, we add Brownian motion to the fluid field. Third, we derive the (virtual) particle's movement and rotation due to the well-known Faxén's correction [12] on the modified fluid field. Finally, we update its position. This can be done for multiple particles simultaneously, as no particle-flow impact is taken into account, i.e. we take the same fluid field to track several virtual particles at the same time. Also, one fluid field is used to derive several variants of the stochastic fluid field disturbed by Brownian motion.

Again, we provide the user with a very coarse on-the-fly visualisation of the flow field augmented by particle position data. As Brownian motion is a local effect, we add Brownian motion only to the fluid data around our virtual particles, only to subregions. As Brownian motion is a small-scale effect, this is done on a very fine grid. As multiple particles are studied simultaneously, multiple regions with a very fine resolution have to be tracked. Both experiments are conducted with our CFD code based on Peano [13, 14]. It works on adaptive Cartesian grids. A local workstation acts as visualisation and postprocessing device while Peano runs on a supercomputer with several thousand cores.

3. The query language

Bandwidth restrictions are particular cumbersome, since, for most of the simulation domain, a very coarse representation of the fluid field is sufficient. However, a simple level of detail approach delivering a first snapshot in a coarse resolution is not well-suited here, as the fluid solver does not know what regions are of interest currently—due to different requirements of the consecutive processing steps and due to the user visualising and studying (parts of) the data. Our postprocessing components hence submit queries defining both the exact region of interest and the data to be studied, as well as the required resolution to the server. The server then sends back answers containing these data, i.e. it does not flood the system with all the simulation data (fire-and-forget semantics) but tailors the results delivered to the requirements.

Algorithm 1 There are three different queries to obtain data from the fluid solver. They define the interface of a `QueryServer`. The right column shows typically answers of the simplest version (without the chemical diffusion) of our fluid solver working on a unit square computational domain.

```
interface QueryServer {
  getSimulationOutline(
    out int       dimension,                    3
    out double[]  boundingBoxOffset,            (0,0,0)
    out double[]  boundingBox                   (1,1,1)
  )
  getScopes(
    out int       numberOfScopes,               2
    out int[]     scopeCardinality,             {1,3}
    out string[]  descriptionOfScope            {pressure,velocity}
  )
  getData(
    in int        scope,
    in double[]   boundingBoxOffset,
    in double[]   boundingBox,
    in int[]      resolution,
    out double[]  data
  );
}
```

There are three types of queries (Algorithm 1): One query makes the server return the outline of the computational domain. One query makes the server return an enumerated list of scopes. A scope corresponds to a solver variable such as velocity, pressure, or boundary type, and it comprises a cardinality and a description. For scalar data such as the pressure, the cardinality is one. For vector data, it equals the spatial dimension of the CFD solver. The third query finally makes the server return one scope's data from a given region with a certain resolution. It is well-defined by its scope identifier, the bounding box given by a hexahedron, and the offset (translation) of this bounding box relative to the coordinate system's origin, as well as the resolution of the data required. The resolution prescribes a regular, equidistant Cartesian grid.

The first two queries are typically asked once per client. With a list of scopes at hand, the client then asks for data (`getData`) in combination with a certain resolution and bounding box describing a Cartesian

grid. The returned floating point arrays' cardinality results from the query resolution times the cardinality of the scope. If the client asks for a vector field such as the velocity for example, the server returns three entries per Cartesian grid point. While it is possible to hard-wire the scopes and their cardinalities, a flexible black-box encapsulation where the client first of all gets a list of all scopes and then accesses the scopes individually provides us with some flexibility. As a result, it is possible to run several CFD simulations simultaneously on one query server. Furthermore, if we augment the CFD simulation by a diffusion, e.g., the client modules interpreting the velocity data or the pressure, respectively, have not to be adopted.

4. Realisation

The query types prescribe an interface of the CFD solver code (Algorithm 1). Our solver hence is a server and the postprocessing components act as clients invoking the remote functions on the server. Queries define regular Cartesian grids with a certain query mesh width. Due to this simplicity, it is straightforward how to decompose a query into several disjoint queries. The counterpart, i.e the merging of queries, is straightforward, too, and describes a binning data composition.

4.1. Multiscale Representation

Our CFD code implements a geometric multiscale solver [14]. Due to an octree-like approach similar to [2], it holds the computational grid in different resolutions. We embed our grid into an axis-aligned bounding box, and then cut the grid recursively into pieces along each coordinate axis. Whether this recursive process is to be stopped is decided for each element in each step independently of its neighbours. Such a process yields adaptive Cartesian grids, and it also yields a cascade of finer and finer tessellations. While the resulting adaptive grid defines our fine grid for the solution of the computational fluid dynamics problem, we use the coarser grids of the construction process as coarse grids of the multigrid solver.

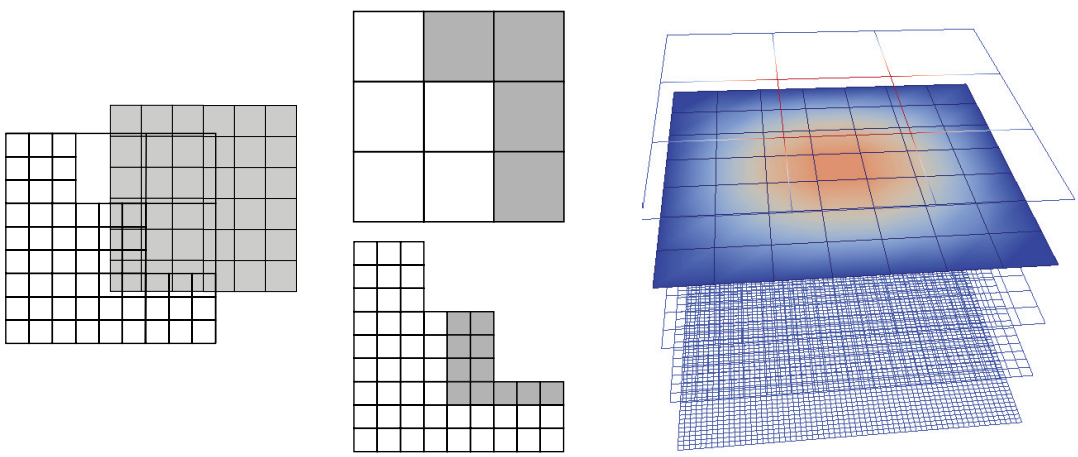

Figure 2: For each query (grayed regular grid, left), the multiscale solver determines which cells of which multiresolution of the adaptive grid are best-suited to answer the query (grayed cells, middle); the query answering process relies on the solution projected to coarser levels (right).

Such a simple construction process on the one hand facilitates the realisation of dynamic time-dependent adaptivity. Whenever a refinement criterion identifies a cell inducing an error that is higher than a given threshold, we refine this cell. The counterpart, i.e. the merge of different cells, equals a coarsening. On the other hand, the visualisation of solutions on such grids requires the application to stream the grid's adjacency and connectivity information for each time step, as the grid itself might change each time step.

Our multigrid solver's realisation follows the idea of a full approximation storage scheme. The important property of this scheme is that the algorithm does not process a correction equation on coarse levels but that it works on a coarsened variant of the solution itself. Whenever the solver derives a solution on any

fine grid, this solution is immediately projected back to all the coarser grids due to a simple induction. As the grids are nested into each other, this projection is a trivial copying—the coarse grid vertices also are fine grid vertices. Our solver process is merged with the query mechanism. After each solver step (a smoothing step throughout a V-cycle for example), it checks the queue of incoming queries. If the fluid solver has received a query throughout the iteration, it takes the coarsest geometric representation where each cell of the grid either contains at most one point of the query grid or belongs to the computational fine grid. In each cell holding a point of the query grid, it then takes the (coarsened) solution and projects this solution onto the query grid (Figure 2).

The queries represent a snapshot of the solution, as the query answering process is not synchronised with the multigrid cycles. If the multiscale solver is processing a finer grid than the query grid, the queries return a coarsened representation of the current solution. To derive this representation, the server however does not have to run through the whole tessellation. It uses the coarsest grid that is well-suited to answer to the query. If the multiscale solver is processing a coarser grid than the query grid, the queries return the snapshot from an old, sufficiently fine resolution. As the solution to our parabolic Navier-Stokes equations does not change rapidly from time step to time step, this asynchronous approach might return data which is not the precise solution to the current time step's problem. Yet, the query server's latency is low.

4.2. Parallelisation

The fluid solver realises a non-overlapping domain decomposition. Domain decomposition and query decomposition go hand in hand: one single node of the supercomputer is the single point of contact (SPoC) and receives the queries Ω_{Query} from the clients. It then decomposes the queries according to the domain decomposition, and each computing node of the supercomputer receives the part of the query grid solely overlapping the domain it is responsible for, i.e. if a node handles a subdomain Ω_i and receives the query, it handles solely to $\Omega_{Query} \cap \Omega_i$. All the query subparts are sent back to the SPoC, merged there, and sent back to the client (Figure 3).

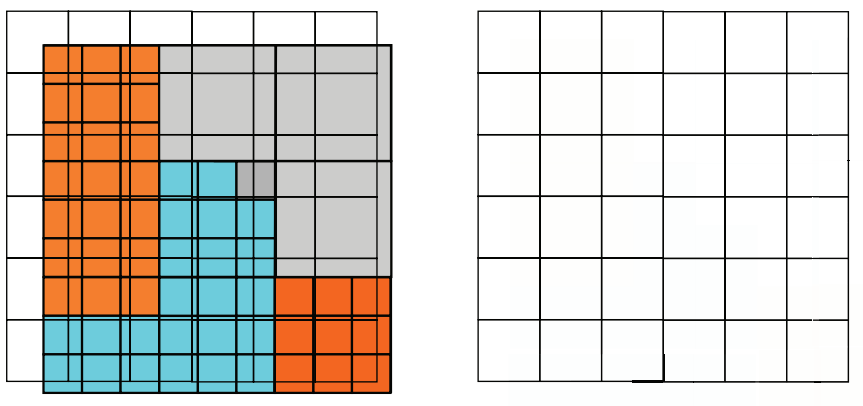

Figure 3: The query defines a regular Cartesian grid, and the CFD solver's domain decomposition (both left) splits up this query into different subqueries accordingly (right).

Several properties stem from this parallelisation approach: First, the parallelisation is hidden from the postprocessing steps due to the single point of contact. The postprocessing steps can use the solver running on a different machine in a black-box manner. Second, nodes handling subdomains that do not intersect with the query grid are not involved in the query answering process. The query thus often affects only a small part of the computing nodes and these computing nodes then share the total communication bandwidth. Third, as the individual nodes apply their multiscale answering mechanism to process the queries and as the grid typically exhibits regions of completely different resolution, some nodes return their subquery grids immediately while others send back their data later, as the data collection process is more complex due to the finer grid. This sending back of data runs in the background of the solver, i.e. is hidden behind the computations, Finally, the mapping from fluid data onto the query grid is parallelised due to the domain

decomposition. It scales. The merge process in turn is very simple as the Ω_i are disjoint and the exchanged data structures are very simple.

We discuss the handling of one query in Section 4.1 and 4.2. However, our realisation accepts multiple queries simultaneously, and they are also processed simultaneously. The answering process however is not synchronised: Queries inducing a coarser mesh grid than other queries might be answered faster and, if this is the case, are sent back earlier. Queries affecting only one single node of the partitioned domain might be answered faster, too, and, if this is the case, are sent back earlier.

5. Results

We tested our query approach for the two different application areas with two supercomputers: the BlueGene/P system Shaheen at the King Abdullah University of Science and Technology (KAUST), and the BlueGene/P system Jugene at the Jülich Supercomputing Centre. Both supercomputers ran our PDE solver framework Peano with its CFD plug-in[2], i.e. they acted as server. For the client submitting the queries, we built a problem solving environment on top of our Eclipse plug-in sciCoDE[3] that picks up the component concept of the common component architecture (CCA) community [15], simplifies it, and provides a graphical user interface to it. It runs on local workstations (Figure 4) and interprets the query server running remotely, the Brownian motion module, the geometry manipulation facilities, and the visualisation codes as individual components. The query interface then describes a component-component connection.

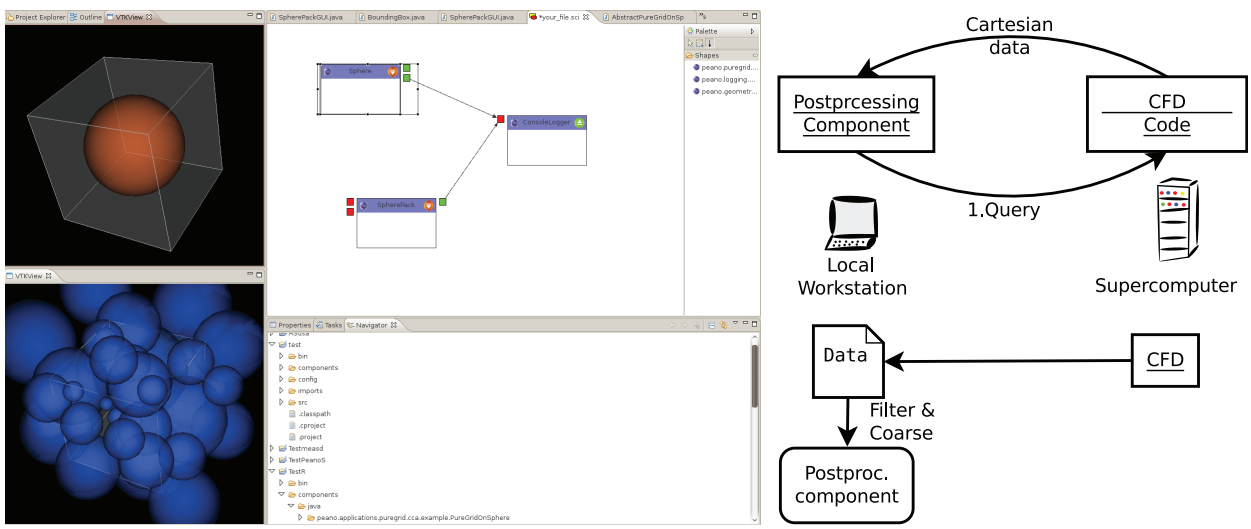

Figure 4: Our problem solving environment is running on a local workstations. It submits queries to the supercomputing acting as query server and interprets the data returned.

For both experiments, we studied the behaviour for several time steps and for comparable resolutions, configurations, and queries. Then, we averaged the results, as both experiments exhibit similar behaviour. The runtime and the memory footprint of the queries are compared to the CFD solver writing one complete solution snapshot. Solution snapshot comprises both the solution data of the variables of interest and the grid connectivity which has to be written together with the simulation data for each time step as the grid changed permanently. Of interest are the velocities and the density of the chemical or the velocities and the pressure, respectively, i.e. $d+1$ doubles per query point. The questions to be answered are "how does the query mechanism fit to the regular data structures and how does it fit to different paradigm of adaptive grids", "how does the query server's behaviour depend on different resolutions of both queries and grids

[2] *www5.in.tum.de/peano*
[3] *www5.in.tum.de/scicode*

Table 1: Comparison of memory footprint of the CFD solver with the memory requirements Mem of three different query combinations and $d = 2$. All memory measurements in MByte, the percentage below is related to the first column of the corresponding row.

CFD Setup			Experiment 1		Experiment 2		Experiment 3	
Grid	#Cells	Mem	Queries	Mem	Queries	Mem	Queries	Mem
regular	$3.5 \cdot 10^5$	81.36	128×32	0.10	128×32	0.12	128×32	0.22
		100%		0.12%	16×64	0.15%	16×64	0.27%
							32×128	
regular	$3.2 \cdot 10^6$	728.99	256×64	0.38	256×64	0.48	256×64	1.24
		100%		0.05%	32×128	0.07%	32×128	0.17%
							64×512	
regular	$2.9 \cdot 10^7$	6570.52	512×128	1.51	512×128	1.90	512×128	4.92
		100%		0.02%	64×256	0.03%	64×256	0.07%
							128×1024	
adaptive	$2.1 \cdot 10^4$	4.88	128×32	0.10	128×32	0.12	128×32	0.22
		100%		2.05%	16×64	2.46%	16×64	4.51%
							32×128	
adaptive	$1.0 \cdot 10^5$	23.10	256×64	0.38	256×64	0.48	256×64	1.24
		100%		1.64%	32×128	2.08%	32×128	5.37%
							64×512	
adaptive	$3.0 \cdot 10^5$	68.70	512×128	1.51	512×128	1.90	512×128	4.92
		100%		2.2%	64×256	2.78%	64×256	7.16%
							128×1024	

Table 2: Table 1 continued for $d = 3$. Again [Mem]=MByte.

CFD Setup			Experiment 1		Experiment 2		Experiment 3	
Grid	#Cells	Mem	Queries	Mem	Queries	Mem	Queries	Mem
regular	$8.6 \cdot 10^7$	$2.7 \cdot 10^4$	$128 \times 32 \times 32$	4.29	$128 \times 32 \times 32$	6.48	$128 \times 32 \times 32$	23.24
		100%		0.01%	$16 \times 64 \times 64$	0.02%	$16 \times 64 \times 64$	0.09%
							$32 \times 128 \times 128$	
regular	$2.3 \cdot 10^9$	$5.4 \cdot 10^5$	$256 \times 64 \times 64$	33.13	$256 \times 64 \times 64$	49.89	$256 \times 64 \times 64$	571.92
		100%		0.01%	$32 \times 128 \times 128$	0.01%	$32 \times 128 \times 128$	0.1%
							$64 \times 512 \times 512$	
adaptive	$1.9 \cdot 10^6$	598.04	$128 \times 32 \times 32$	4.29	$128 \times 32 \times 32$	6.48	$128 \times 32 \times 32$	23.24
		100%		0.72%	$16 \times 64 \times 64$	1.08%	$16 \times 64 \times 64$	3.89%
							$32 \times 128 \times 128$	
adaptive	$1.5 \cdot 10^7$	4576.89	$256 \times 64 \times 64$	33.13	$256 \times 64 \times 64$	49.89	$256 \times 64 \times 64$	571.92
		100%		0.72%	$32 \times 128 \times 128$	1.09%	$32 \times 128 \times 128$	12.5%
							$64 \times 512 \times 512$	

in terms of memory", "how does it react to multiple queries", and "how does the answering fit to the parallel solver". The tables give figures for regular and adaptive grids, for different simulation grid sizes, for different query sizes, and for different numbers of queries sent simultaneously to the query server. The queries correspond to the level of detail study described in the motivation: First, we analyse the global fluid behaviour on a rather coarse grid. Then, we zoom into the regions of interest with one or two finer queries.

The experiments in Table 1 and Table 2 reveal that the query mechanism reduces the memory footprint of the data to be transferred back to the problem solving environment by orders of magnitude. The reduction effect is more significant for regular grids, and it is more significant for $d = 3$ than for $d = 2$: As the solver uses an adaptive Cartesian grid that is adopted to the solver's needs autonomously, increasing the simulation's resolution in particular affects the boundary resolution. The solver identifies that the biggest

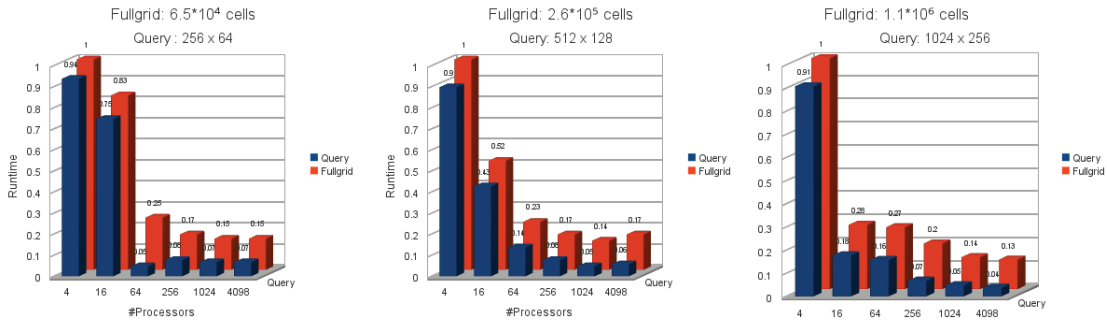

Figure 5: Runtimes of the regular grid CFD solver ($d = 2$) for six different core numbers. The runtimes are normalised with respect to the single node runtime streaming the whole grid.

errors stem from an insufficient boundary resolution and, hence, it refines there more aggressively than it does in other regions of the computational domain. Our query grids however are independent of the boundary resolution and the answering process is independent of the additional effort the CFD solver has to spend to tackle the boundary.

The experiments in Figure 5 study one iteration of the CFD code working on a regular grid that streams either the whole grid or the answer to one query to the postprocessing components. All runtimes are normalised with respect to the single node performance streaming of the whole grid. The figures reveal that the approach scales, i.e. does not harm the parallel efficiency of the original CFD code. This is on the one hand due to the fact that the query answering for the multiscale algorithm runs in the background of the actual solver. It is hidden behind the computation completely, if the problem size per computing node is sufficiently big. On the other hand, it is also due to the fact that the data merge for the queries can run in parallel to the computations on the SPoC—as soon as query data arrives at the SPoC, the server mergers it into the query answer data structure. If we stream the whole experiment data, this merge phase has to take place after the computing iteration has finished introducing a sequential postprocessing phase.

6. Conclusion and outlook

In this paper, we introduce a very simple data exchange interface consisting of only three operations. The underlying communication paradigm switches from a fire-and-forget to a demand-driven approach: the simulation code does not deliver data permanently, but delivers solely data that is explicitly requested. A request comprises not only the type of the data but also its bounding box and the required resolution. Such a communication scheme coins the layout of computational steering and interactive visualisation simulation environments: they are not data-flow driven anymore, but rely on the exchange of data requirements.

Our realisation of the query server replying to queries reveals that in particular parallel multiscale solvers benefit from the demand-driven paradigm. With a multiscale representation of the solution at hand, they deliver query answers fast as they systematically pick out an appropriate grid resolution that fits to the query's resolution. With a domain decomposition of the solution at hand, computing nodes of the query server answer to those parts of the query where they already hold the data of the corresponding subdomain. These two advantages lead to a scaling query server with a low latency. Besides the runtime, the simplicity of the queries implies that the memory overhead for the query data is very low. There is no connectivity or adjacency information to be stored. Furthermore, aspects like the domain decomposition as well as the adaptive structure of the simulation grid are hidden from the postprocessing units.

Future work in our group comprises some natural extension points: First, the requirement-driven communication paradigm has to be shown to be of value for further real-world, massively parallel applications—in terms of usability, in terms of performance, and in terms of data volume. Second, while the single point of contact paradigm makes the coupling of different applications running with different decomposition schemes on different computers simple, it induces a critical, sequential communication point and, almost for sure, will

slow down the overall application as soon as the postprocessing components run in parallel, too. However, with different parallel codes coupled due to queries, query server nodes could communicate to requesting client nodes directly. Third, our implementation does not use any IO forwarding yet. However, the approach is well-suited for IO forwarding and can benefit from it. Finally, our work does not take into account any in-situ processing and data compression yet. Particularly promising is the combination of our query mechanism with database alike select statements (select for example only values bigger than a threshold) as well as reduce and simple data manipulation operations (such as normalisation of all query data, e.g.)—an approach similar to on-the-fly event processing for discrete data. Such queries can be deployed to IO nodes or processed in-situ and, once again, reduce the communication ressource requirements. We end up with more science per transferred byte.

Acknowledgements

This publication is partially based on work supported by Award No. UK-c0020, made by the King Abdullah University of Science and Technology (KAUST). We furthermore thank the DEISA Consortium (`www.deisa.eu`), co-funded through the EU FP6 project RI-031513 and the FP7 project RI-222919, for support within the DEISA Extreme Computing Initiative. Additional thanks are due to Mihaela Mihai and Denis Jarema for implementing several software components used in this work.

References

[1] J. X. Chen, D. Rine, H. D. Simon, Theme Editors' Introduction: Advancing Interactive Visualization and Computational Steering, IEEE Comput. Sci. Eng. 3 (1996) 13–17.
[2] T. Tu, H. Yu, J. Bielak, O. Ghattas, J. C. Lopez, K.-L. Ma, D. R. O'Hallaron, L. Ramirez-Guzman, N. Stone, R. Taborda-Rios, J. Urbanic, Remote runtime steering of integrated terascale simulation and visualization, in: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, SC '06, ACM, New York, NY, USA, 2006.
[3] K. Bürger, J. Schneider, P. Kondratieva, J. Krüger, R. Westermann, Interactive Visual Exploration of Instationary 3D-Flows, in: EuroVis—Eurographics/IEEE VGTC Symposium on Visualization, 2007, pp. 251–258.
[4] J. P. Ahrens, J. Woodring, D. E. DeMarle, J. Patchett, M. Maltrud, Interactive remote large-scale data visualization via prioritized multi-resolution streaming, in: Proceedings of the 2009 Workshop on Ultrascale Visualization, UltraVis '09, ACM, New York, NY, USA, 2009, pp. 1–10.
[5] H.-J. Bungartz, W. Eckhardt, T. Weinzierl, C. Zenger, A Precompiler to Reduce the Memory Footprint of Multiscale PDE Solvers in C++, Future Generation Computer Systems 26 (1) (2010) 175–182.
[6] K. Stockinger, J. Shalf, K. Wu, E. W. Bethel, Query-Driven Visualization of Large Data Sets, in: Proceedings of IEEE Visualization 2005, IEEE Computer Society Press, 2005, pp. 167–174, lBNL-57511.
[7] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. E. Papka, R. Ross, K. Yoshii, Accelerating I/O Forwarding in IBM Blue Gene/P Systems, in: SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society, Washington, DC, USA, 2010.
[8] C. R. Johnson, R. Ross, S. Ahern, J. Ahrens, W. Bethel, K. L. Ma, M. Papka, J. van Rosendale, H. W. Shen, J. Thomas, Visualization and Knowledge Discovery: Report from the DOE/ASCR Workshop on Visual Analysis and Data Exploration at Extreme Scale (2007).
[9] A. Modi, L. N. Long, P. E. Plassmann, Real-time visualization of wake-vortex simulations using computational steering and Beowulf clusters, in: Proceedings of the 5th international conference on High performance computing for computational science, VECPAR'02, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 464–478.
[10] G. H. Weber, S. Ahern, E. W. Bethel, S. Borovikov, H. R. Childs, E. Deines, C. Garth, H. Hagen, B. Hamann, K. I. Joy, D. Martin, J. Meredith, P. Pugmire, D. Pugmire, O. Rübel, B. V. Straalen, K. Wu, Recent Advances in VisIt: AMR Streamlines and Query-Driven Visualization, in: Numerical Modeling of Space Plasma Flows: Astronum-2009 (Astronomical Society of the Pacific Conference Series), Vol. 429, 2010, pp. 329–334, lBNL-3185E.
[11] M. Brenk, H.-J. Bungartz, M. Mehl, I. L. Muntean, T. Neckel, T. Weinzierl, Numerical Simulation of Particle Transport in a Drift Ratchet, SIAM Journal of Scientific Computing 30 (6) (2008) 2777–2798.
[12] M. Schindler, Free-Surface Microflows and Particle Transport, Ph.D. thesis, Universität Augsburg, Universitätsstr. 22, 86159 Augsburg (2006).
[13] H.-J. Bungartz, M. Mehl, T. Neckel, T. Weinzierl, The PDE framework Peano applied to fluid dynamics: An efficient implementation of a parallel multiscale fluid dynamics solver on octree-like adaptive Cartesian grids, Computational Mechanics 46 (1) (2010) 103–114, published online.
[14] T. Weinzierl, A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids, Verlag Dr. Hut, 2009.
[15] B. A. Allan, R. Armstrong, D. E. Bernholdt, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. Mclnnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, S. Zhou, A Component Architecture for High-Performance Scientific Computing, Int. J. High Perform. Comput. Appl. 20 (2006) 163–202.